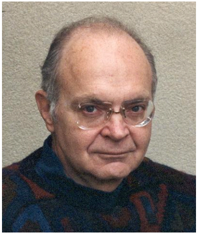
Design and Analysis of Algorithms
Introduction to Algorithms

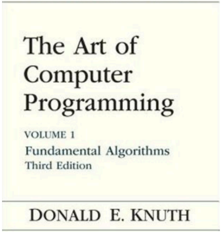
1. A Taste of Algorithm Design
   - Return on Investment (ROI) Problem
   - Single Machine Scheduling (SMS) Problem
2. A Taste of Algorithm Analysis
   - Sorting Problem
3. A Taste of Complexity Theory
   - Travelling Salesman Problem
   - Knapsack Problem

What is Algorithm?

"*An algorithm is a finite, definite, effective procedure, with some input and some output.*"

– Donald Knuth





1. A Taste of Algorithm Design
   - Return on Investment (ROI) Problem
   - Single Machine Scheduling (SMS) Problem
2. A Taste of Algorithm Analysis
   - Sorting Problem
3. A Taste of Complexity Theory
   - Travelling Salesman Problem
   - Knapsack Problem

Return on Investment (ROI) Problem

Problem. m coins to invest n projects.

- profit function $f_i(x)$ denotes the return on investing project i with x coins, $i = 1, 2, \ldots, n$.

How to maximize the overall return?

Instance example: 5 coins, 4 projects:

x	$f_1(x)$	$f_2(x)$	$f_3(x)$	$f_4(x)$
0	0	0	0	0
1	11	0	2	20
2	12	5	10	21
3	13	10	30	22
4	14	15	32	23
5	15	20	40	24

Modeling

Input. n, m, $f_i(x)$, $i \in [n]$, $x \in \{0, \ldots, m\}$

Solution. vector $\langle x_1, x_2, \ldots, x_n \rangle$, x_i is the num of coins invested on project i satisfying:

$$\text{objective function:} \max \sum_{i=1}^{n} f_i(x_i)$$

$$\text{constraints:} \sum_{i=1}^{n} x_i = m, x_i \in \{0, \ldots, m\}$$

Brute-Force Algorithm: Universal Algorithm for All Problems

Definition 1 (Brute-Force Algorithm)

A programming style that does not use any shortcuts to improve performance, but instead relies on sheer computing power to try all possibilities until the solution to a problem is found.

Brute-Force Algorithm: Universal Algorithm for All Problems

Definition 1 (Brute-Force Algorithm)

A programming style that does not use any shortcuts to improve performance, but instead relies on sheer computing power to try all possibilities until the solution to a problem is found.

$\forall$ n-dimension vector $\langle x_1, x_2, \ldots, x_n \rangle$ satisfying

$$x_1 + x_2 + \cdots + x_n = m, x_i \in \{0, \ldots, m\}$$

compute the sum of return

$$f_1(x_1) + f_2(x_2) + \cdots + f_n(x_n)$$

find the solution with highest return

Example

x	$f_1(x)$	$f_2(x)$	$f_3(x)$	$f_4(x)$
0	0	0	0	0
1	11	0	2	20
2	12	5	10	21
3	13	10	30	22
4	14	15	32	23
5	15	20	40	24

$$x_1 + x_2 + x_3 + x_4 = 5$$

$s_1 = \langle 0, 0, 0, 5 \rangle,\ v(s_1) = 24$

$s_2 = \langle 0, 0, 1, 4 \rangle,\ v(s_2) = 25$

$s_3 = \langle 0, 0, 2, 3 \rangle,\ v(s_3) = 32$

$\ldots$

$s_{56} = \langle 5, 0, 0, 0 \rangle,\ v(s_{56}) = 15$

Solution: $s = \langle 1, 0, 3, 1 \rangle$

Highest return: $11 + 30 + 20 = 61$

Efficiency of Brute-Force Algorithm

Each possible solution vector is a non-negative integer solution of equation

$$x_1 + x_2 + \cdots + x_n = m$$

Efficiency of Brute-Force Algorithm

Each possible solution vector is a non-negative integer solution of equation

$$x_1 + x_2 + \cdots + x_n = m$$

How to estimate the number of possible $\langle x_1, x_2, \ldots, x_n \rangle$

- solution can be expressed as 0-1 sequence with the following format: $\#\ 1 = m$, $\#\ 0 = n-1$

$$\underbrace{1\ \ldots\ 1}_{x_1}\ 0\ \underbrace{1\ \ldots\ 1}_{x_2}\ 0\ \ldots\ 0\ \underbrace{1\ \ldots\ 1}_{x_n}$$

$n = 4$, $m = 7$

candidate solution $\langle 1, 2, 3, 1 \rangle$ corresponds to:

$$1\ 0\ 1\ 1\ 0\ 1\ 1\ 1\ 0\ 1$$

Efficiency of Brute-Force Algorithm

The number of such sequences is an exponential function of input size

$$
\begin{aligned}
C(m+n-1, n-1) &= \frac{(m+n-1)!}{m!(n-1)!} \\
&= \Omega((1+\epsilon)^{m+n-1})
\end{aligned}
$$

Efficiency of Brute-Force Algorithm

The number of such sequences is an exponential function of input size

$$
\begin{aligned}
C(m+n-1, n-1) &= \frac{(m+n-1)!}{m!(n-1)!} \\
&= \Omega((1+\epsilon)^{m+n-1})
\end{aligned}
$$

An alternative reasoning: calculate the number of positive integer solutions

$$y_1 + y_2 + \cdots + y_n = m + n$$

Efficiency of Brute-Force Algorithm

The number of such sequences is an exponential function of input size

$$\begin{aligned} C(m+n-1, n-1) &= \frac{(m+n-1)!}{m!(n-1)!} \\ &= \Omega((1+\epsilon)^{m+n-1}) \end{aligned}$$

An alternative reasoning: calculate the number of positive integer solutions

$$y_1 + y_2 + \cdots + y_n = m + n$$

Brute-force algorithm is easy to design when the solution space is enumerable, and always correct, but not efficient when the solution space is huge.

In most time, we need to design "smart" algorithm.

Single Machine Scheduling Problem

Problem. n tasks, each task i requires time t_i to process (without waiting), refereed to minimum processing time. We have to assign n tasks on a single machine.

- flowtime of task i: $start_i = 0$, $end_i - start_i \geq t_i$

Performance goal. find an assignment such that the total flowtime of all n tasks is shortest.

Modeling

Input.

- task set: $S = \{1, 2, \ldots, n\}$
- processing time of task j: $t_j \in \mathbb{Z}^+$, $j \in [n]$

Output. Schedule I, a permutation of S, i.e., $(i_1, i_2, \ldots, i_n)$

Objective function. the flowtime of I:

$$t(I) = \sum_{k=1}^{n} (n - k + 1) t_{i_k}$$

Solution. I^* — minimize $t(I^*)$

$$t(I^*) = \min\{t(I) \mid I \in \text{Permutation}(S)\}$$

Solve: Greedy Algorithm

Greedy algorithm is a kind of heuristic algorithms

- originated from your intuition
- follow your heart

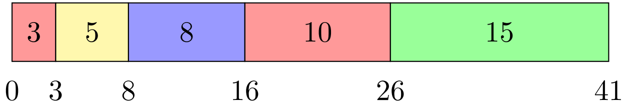
Strategy. shortest processing time (SPT) first

Algorithm. sort the processing time in an increasing order, then process them sequentially

Concrete Instance

- task set $S = \{1, 2, 3, 4, 5\}$
- minimum processing time: $t_1 = 3$, $t_2 = 8$, $t_3 = 5$, $t_4 = 10$, $t_5 = 15$

sort $(3, 8, 5, 10, 15)$ in an increasing order $\rightsquigarrow$ Solution: $1, 3, 2, 4, 5$



overall flowtime

$$\begin{aligned} t &= 3 + (3+5) + (3+5+8) + (3+5+8+10) \\ &\quad + (3+5+8+10+15) \\ &= 3 \times 5 + 5 \times 4 + 8 \times 3 + 10 \times 2 + 15 \\ &= 94 \end{aligned}$$

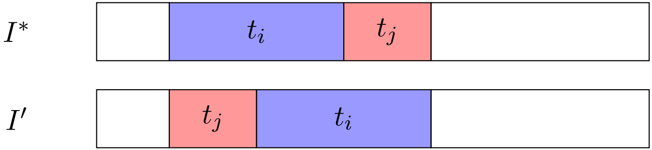
Proof of Correctness

Correctness. We have to ensure greedy algorithm yields the optimal solutions for *all instances*

Proof of Correctness

Correctness. We have to ensure greedy algorithm yields the optimal solutions for *all instances*

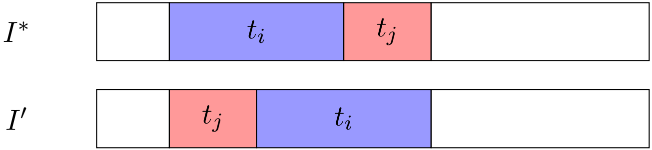
Proof. If not $\Rightarrow \exists$ optimal schedule I^* with at least one reverse order, i.e., task i and j are adjacent but $t_i > t_j$. Switch task i and j in $I^* \rightsquigarrow$ schedule I'



Proof of Correctness

Correctness. We have to ensure greedy algorithm yields the optimal solutions for *all instances*

Proof. If not $\Rightarrow \exists$ optimal schedule I^* with at least one reverse order, i.e., task i and j are adjacent but $t_i > t_j$. Switch task i and j in $I^* \rightsquigarrow$ schedule I'



flowtime comparison: $t(I') - t(I^*) = t_j - t_i < 0 \Rightarrow$ contradicts to the optimal property of I^*

Heuristics is not always Correct

Counterexample.

Knapsack problem: four items need to insert into a knapsack, with values and weights as below:

label	a	b	c	d
weight w_i	3	4	5	2
value v_i	7	9	9	2

the knapsack weight limit is 6.

How to choose items to maximize the total values in the backpack?

Failure of Greedy Algorithm

Greedy strategy. highest value-weight ratio comes first, with weight limit 6

- sort v_i/w_i in a descending order: a, b, c, d

$$\boxed{\frac{7}{3}} > \frac{9}{4} > \frac{9}{5} > \boxed{\frac{2}{2}}$$

greedy solution: $\{a, d\}$, weight $= 5$, value $= 9$

better solution: $\{b, d\}$, weight $= 6$, value $= 11$

Summary: Steps of Algorithm Design

1. Modeling. give formal description of input, output and objective function

Summary: Steps of Algorithm Design

1. Modeling. give formal description of input, output and objective function
2. Design. Choose what algorithms? How to describe it?

Summary: Steps of Algorithm Design

1. Modeling. give formal description of input, output and objective function
2. Design. Choose what algorithms? How to describe it?
3. Prove. Is the algorithm correct: yielding optimal solution for all instances.
    - If so, how to prove it?
    - If not, can you find an counterexample?

Summary: Steps of Algorithm Design

1. Modeling. give formal description of input, output and objective function
2. Design. Choose what algorithms? How to describe it?
3. Prove. Is the algorithm correct: yielding optimal solution for all instances.
   - If so, how to prove it?
   - If not, can you find an counterexample?
4. Analysis. efficiency: time and space

1. A Taste of Algorithm Design
   - Return on Investment (ROI) Problem
   - Single Machine Scheduling (SMS) Problem
2. A Taste of Algorithm Analysis
   - Sorting Problem
3. A Taste of Complexity Theory
   - Travelling Salesman Problem
   - Knapsack Problem

Insertion Algorithm

Insertion sort iterates, consuming one input element each iteration, and growing a sorted output list.

- At each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there.
- It repeats until no input elements remain.

input	5	7	1	3	6	2	4
middle state	1	3	5	6	7	2	4
after inserting 2	1	2	3	5	6	7	4

Demo of Insertion Sort

input							
input	5	7	1	3	6	2	4

Demo of Insertion Sort

input	5	7	1	3	6	2	4
beginning	5	7	1	3	6	2	4

Demo of Insertion Sort

input	5	7	1	3	6	2	4
beginning	5	7	1	3	6	2	4
insert 7	5	7	1	3	6	2	4

Demo of Insertion Sort

input	5	7	1	3	6	2	4
beginning	5	7	1	3	6	2	4
insert 7	5	7	1	3	6	2	4
insert 1	1	5	7	3	6	2	4

Demo of Insertion Sort

input	5	7	1	3	6	2	4
beginning	5	7	1	3	6	2	4
insert 7	5	7	1	3	6	2	4
insert 1	1	5	7	3	6	2	4
insert 3	1	3	5	7	6	2	4

Demo of Insertion Sort

input	5	7	1	3	6	2	4
beginning	5	7	1	3	6	2	4
insert 7	5	7	1	3	6	2	4
insert 1	1	5	7	3	6	2	4
insert 3	1	3	5	7	6	2	4
insert 6	1	3	5	6	7	2	4

Demo of Insertion Sort

input	5	7	1	3	6	2	4
beginning	5	7	1	3	6	2	4
insert 7	5	7	1	3	6	2	4
insert 1	1	5	7	3	6	2	4
insert 3	1	3	5	7	6	2	4
insert 6	1	3	5	6	7	2	4
insert 2	1	2	3	5	6	7	4

Demo of Insertion Sort

input	5	7	1	3	6	2	4
beginning	5	7	1	3	6	2	4
insert 7	5	7	1	3	6	2	4
insert 1	1	5	7	3	6	2	4
insert 3	1	3	5	7	6	2	4
insert 6	1	3	5	6	7	2	4
insert 2	1	2	3	5	6	7	4
insert 4	1	2	3	4	5	6	7

Analysis of Insertion Sort

Complexity analysis

- worst-case: $O(n^2)$ comparison and swap
- best-case: $O(n)$ comparison and $O(1)$ swap
- average-case: $O(n^2)$ comparison and swap

replace array with linked list: reduce swap operation in each round to constant time

Analysis of Insertion Sort

Complexity analysis

- worst-case: $O(n^2)$ comparison and swap
- best-case: $O(n)$ comparison and $O(1)$ swap
- average-case: $O(n^2)$ comparison and swap

replace array with linked list: reduce swap operation in each round to constant time

---

Advantages:

- simple: Jon Bentley shows a three-line C version
- adaptive: efficient for data sets that are already substantially sorted
- stable: does not change the relative order of elements with equal keys
- in-place: only require constant additional memory
- online: can sort a data set as it receives it

Bubble Sort

Bubble sort: pass through the list — compares adjacent elements and swaps them if they are in the wrong order.

- the pass is repeated until the list is sorted
- named for the way smaller or larger elements "bubble" to the top of the list (another name is sinking sort)

before pass

5	1	6	2	8	3	4	7

one pass

1	5	2	6	3	4	7	8

Demo of Bubble Sort

input	5	8	1	3	6	2	4	7

Demo of Bubble Sort

input	5	8	1	3	6	2	4	7
pass 1	5	1	3	6	2	4	7	8

Demo of Bubble Sort

input	5	8	1	3	6	2	4	7
pass 1	5	1	3	6	2	4	7	8
pass 2	1	3	5	2	4	6	7	8

Demo of Bubble Sort

input	5	8	1	3	6	2	4	7
pass 1	5	1	3	6	2	4	7	8
pass 2	1	3	5	2	4	6	7	8
pass 3	1	3	2	4	5	6	7	8

Demo of Bubble Sort

input	5	8	1	3	6	2	4	7
pass 1	5	1	3	6	2	4	7	8
pass 2	1	3	5	2	4	6	7	8
pass 3	1	3	2	4	5	6	7	8
pass 4	1	2	3	4	5	6	7	8

Demo of Bubble Sort

input	5	8	1	3	6	2	4	7
pass 1	5	1	3	6	2	4	7	8
pass 2	1	3	5	2	4	6	7	8
pass 3	1	3	2	4	5	6	7	8
pass 4	1	2	3	4	5	6	7	8
pass 5	1	2	3	4	5	6	7	8

Analysis of Bubble Sort

Complexity analysis

- worst-case: $O(n^2)$ comparison and swap
- best-case: $O(n)$ comparison and $O(1)$ swap
- average-case: $O(n^2)$ comparison and swap

Analysis of Bubble Sort

Complexity analysis

- worst-case: $O(n^2)$ comparison and swap
- best-case: $O(n)$ comparison and $O(1)$ swap
- average-case: $O(n^2)$ comparison and swap

Advantages. Simple and stable

Disadvantages. Inefficient, only for education purpose

Quick Sort

Quicksort is a divide-and-conquer algorithm:

1. Pick an element, called a pivot, from the array.
2. Partitioning: reorder the array to a low sub-array (values smaller than the pivot) and a high sub-array (values larger than the pivot), equal values can go either way (or stay in the middle). After this partitioning, the pivot is in its final position.

Recursively apply the above steps to the sub-arrays.

Quick Sort

Quicksort is a divide-and-conquer algorithm:

1. Pick an element, called a pivot, from the array.
2. Partitioning: reorder the array to a low sub-array (values smaller than the pivot) and a high sub-array (values larger than the pivot), equal values can go either way (or stay in the middle). After this partitioning, the pivot is in its final position.

Recursively apply the above steps to the sub-arrays.

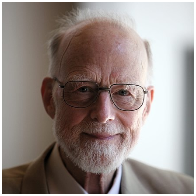


Figure: Tony Hoare

invent in 1959 in Moscow State University Soviet Union, where he studied machine translation under Andrey Kolmogorov

Most significant work: Quicksort and Quickselect, Hoare logic, Communicating Sequential Processes (CSP) for concurrent processes

One Recursive Round of Quick Sort

How to partition: two pointers trick

input	5	8	1	3	6	2	4	7

One Recursive Round of Quick Sort

How to partition: two pointers trick

input	5	8	1	3	6	2	4	7
1st swap	5	4	1	3	6	2	8	7

One Recursive Round of Quick Sort

How to partition: two pointers trick

input	5	8	1	3	6	2	4	7
1st swap	5	4	1	3	6	2	8	7
2nd swap	5	4	1	3	2	6	8	7

cross happens

One Recursive Round of Quick Sort

How to partition: two pointers trick

input	5	8	1	3	6	2	4	7
1st swap	5	4	1	3	6	2	8	7
2nd swap	5	4	1	3	2	6	8	7
					cross happens			
partition	2	4	1	3	5	6	8	7

One Recursive Round of Quick Sort

How to partition: two pointers trick

input	5	8	1	3	6	2	4	7
1st swap	5	4	1	3	6	2	8	7
2nd swap	5	4	1	3	2	6	8	7
					cross happens			
partition	2	4	1	3	5	6	8	7
sub problem	4	1	3	2	5	6	8	7

Analysis of Quick Sort

Complexity analysis

Analysis of Quick Sort

Complexity analysis

- worst-case: $O(n^2)$ comparison and swap (think about when?)

Analysis of Quick Sort

Complexity analysis

- worst-case: $O(n^2)$ comparison and swap (think about when?)

  totally ordered or unordered

Analysis of Quick Sort

Complexity analysis

- worst-case: $O(n^2)$ comparison and swap (think about when?)

  totally ordered or unordered

- best-case: $O(n \log n)$ comparison and $O(1)$ swap
- average-case: $O(n \log n)$ comparison and swap

Analysis of Quick Sort

Complexity analysis

- worst-case: $O(n^2)$ comparison and swap (think about when?)

  totally ordered or unordered

- best-case: $O(n \log n)$ comparison and $O(1)$ swap
- average-case: $O(n \log n)$ comparison and swap

Advantages

- quick: gained widespread adoption, e.g., (i) in Unix as the default library sort subroutine; (ii) it lent its name to the C standard library subroutine `qsort`; (iii) in the reference implementation of Java.

Properties

- non-stable
- pivot-choice affect performance

Merge Sort

Merge sort is also a divide-and-conquer algorithm:

- divide the unsorted list into n sublists, each containing one element (a list of one element is considered sorted).
- repeatedly merge sublists to produce new sorted sublists until there is only one sublist remaining. (this will be the sorted list.)

Canonical case $n = 2^k$

Figure: John von Neumann

Demo of Merge Sort

input								
input	5	8	1	3	6	2	4	7

Demo of Merge Sort

input	5	8	1	3	6	2	4	7
1st merge	5	8	1	3	2	6	4	7

Demo of Merge Sort

input	5	8	1	3	6	2	4	7
1st merge	5	8	1	3	2	6	4	7
2nd merge	1	3	5	8	2	4	6	7

Demo of Merge Sort

input	5	8	1	3	6	2	4	7
1st merge	5	8	1	3	2	6	4	7
2nd merge	1	3	5	8	2	4	6	7
3rd merge	1	2	3	4	5	6	7	8

Analysis of Merge Sort

Complexity analysis

- worst-case, best-case, average-case: $O(n \log n)$ comparison
- space: $O(n)$ total with $O(n)$ auxiliary (not in-place)

Advantages

- quick: (i) Linux kernel for linked list; (ii) Android platform; (iii) default sort algorithm in python and Java

Property

- stable

Comparisons Among Sorting Algorithms

Algorithm	worst case	best case	average case	stable
insertion sort	$O(n^2)$	$O(n)$	$O(n^2)$	yes
bubble sort	$O(n^2)$	$O(n)$	$O(n^2)$	yes
quick sort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$	no
merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	yes

Complexity Analysis

Which algorithm performs best?
How to evaluate it?

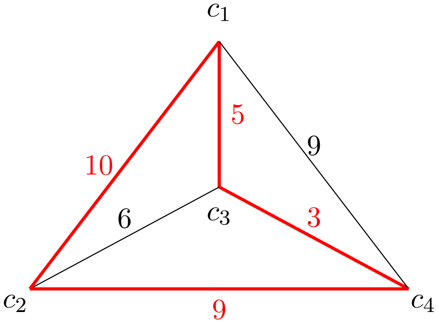
Can we find better sorting algorithm?

n^2	insertion sort bubble sort quick sort
$n \log n$	merge sort
?	better lower bound

1. A Taste of Algorithm Design
    - Return on Investment (ROI) Problem
    - Single Machine Scheduling (SMS) Problem
2. A Taste of Algorithm Analysis
    - Sorting Problem
3. A Taste of Complexity Theory
    - Travelling Salesman Problem
    - Knapsack Problem

Travelling Salesman Problem (TSP)

Problem. Given n cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?



Formalization

Input. Finite set of cities $C = \{c_1, c_2, \ldots, c_n\}$, distance $d(c_i, c_j) = d(c_j, c_i) \in \mathbb{Z}^+$, $1 \leq i < j \leq n$.

Solution. A permutation of $1, 2, \ldots, n$, a.k.a. $k_1, k_2, \ldots, k_n$ such that:

$$\min \left\{ \sum_{i=1}^{n-1} d(c_{k_i}, c_{k_{i+1}}) + d(c_{k_n}, c_{k_1}) \right\}$$

Formalization

Input. Finite set of cities $C = \{c_1, c_2, \ldots, c_n\}$, distance $d(c_i, c_j) = d(c_j, c_i) \in \mathbb{Z}^+$, $1 \leq i < j \leq n$.

Solution. A permutation of $1, 2, \ldots, n$, a.k.a. $k_1, k_2, \ldots, k_n$ such that:

$$\min \left\{ \sum_{i=1}^{n-1} d(c_{k_i}, c_{k_{i+1}}) + d(c_{k_n}, c_{k_1}) \right\}$$

Can the objective function be simpler?

- use modular n expression — $0, 1, \ldots, n-1$

$$\min \left\{ \sum_{i=0}^{n-1} d(c_{k_i}, c_{k_{i+1}}) \right\}$$

About TSP

TSP (first formulated in 1930) is the most intensively studied problems $\mathcal{NP}$-hard problem in combinatorial optimization and theoretical computer science.

About TSP

TSP (first formulated in 1930) is the most intensively studied problems $\mathcal{NP}$-hard problem in combinatorial optimization and theoretical computer science.

TSP is used as a benchmark for many optimization methods. Though TSP is computationally difficult, many heuristics and approximated algorithms are known.

- some instances with tens of thousands of cities can be solved completely
- even problems with millions of cities can be approximated within a small fraction of 1%.

About TSP

TSP (first formulated in 1930) is the most intensively studied problems $\mathcal{NP}$-hard problem in combinatorial optimization and theoretical computer science.

TSP is used as a benchmark for many optimization methods. Though TSP is computationally difficult, many heuristics and approximated algorithms are known.

- some instances with tens of thousands of cities can be solved completely
- even problems with millions of cities can be approximated within a small fraction of 1%.

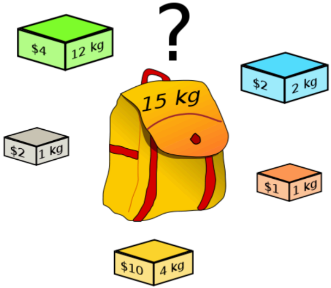
TSP has several applications

- in its purest formulation: planning, logistics, and the manufacture of microchips
- slightly modified: DNA sequencing

Knapsack Problem

Given n items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit W and the total value is as large as possible.

- name: someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items
- 0-1 variant: for each item, include or not



Formalization

Solution. vector $\langle x_1, x_2, \ldots, x_n \rangle$ over $\{0,1\}^n$, $x_i = 1$ iff item i is included

$$\text{objective function: } \max \sum_{i=1}^{n} v_i x_i$$

$$\text{constraint: } \sum_{i=1}^{n} w_i x_i \leq W, x_i \in \{0,1\}, i \in [n]$$

About Knapsack Problem

Knapsack (since 1897) often arises in resource allocation where the decision makers have to choose from a set of *non-divisible* projects or tasks under a fixed budget or time constraint, respectively. It is $\mathcal{NP}$-complete problem.

About Knapsack Problem

Knapsack (since 1897) often arises in resource allocation where the decision makers have to choose from a set of *non-divisible* projects or tasks under a fixed budget or time constraint, respectively. It is $\mathcal{NP}$-complete problem.

Hardness of the knapsack problem depends on the form of the input.

- one theme in research is to identify "hard" instances: identify what properties of instances might make them more amenable than their worst-case $\mathcal{NP}$-complete hardness suggests
- application in public-key cryptography systems, e.g., the Merkle-Hellman knapsack cryptosystem.

About Knapsack Problem

Knapsack (since 1897) often arises in resource allocation where the decision makers have to choose from a set of *non-divisible* projects or tasks under a fixed budget or time constraint, respectively. It is $\mathcal{NP}$-complete problem.

Hardness of the knapsack problem depends on the form of the input.

- one theme in research is to identify "hard" instances: identify what properties of instances might make them more amenable than their worst-case $\mathcal{NP}$-complete hardness suggests
- application in public-key cryptography systems, e.g., the Merkle-Hellman knapsack cryptosystem.

The basic problem is a one-dimensional (constraint) knapsack problem

- a multiple constrained problem could consider both the weight and volume of knapsack

$\mathcal{NP}$-hard Problem

$\mathcal{NP}$-hardness (non-deterministic polynomial-time hardness) is a class of problems that are

- informally "at least as hard as the hardest problems in $\mathcal{NP}$"
- an efficient algorithm for a $\mathcal{NP}$-hard problem implies efficient algorithms for all $\mathcal{NP}$ problem

No "efficient" algorithms found yet:

- complexity of known algorithm are at least exponential function on input size
- no one can prove the "non-existence" of efficient algorithms for those problems

Thousands of $\mathcal{NP}$-hard problems, widely spreads in all areas.

Summary

The significance of algorithm

Algorithm evaluation criteria

- Efficient: low time complexity & space complexity
- Correct: yield optimal solution for all instances

The Scope of Algorithm

- Design technique (exemplified by SMS and ROI)
  - modeling $\rightsquigarrow$ find an algorithm
  - proof $\rightsquigarrow$ prove the correctness
- Complexity analysis (exemplified by sorting problem)
  - calculate the number of basic operations
- Complexity theory (TSP and Knapsack)
  - complexity classification